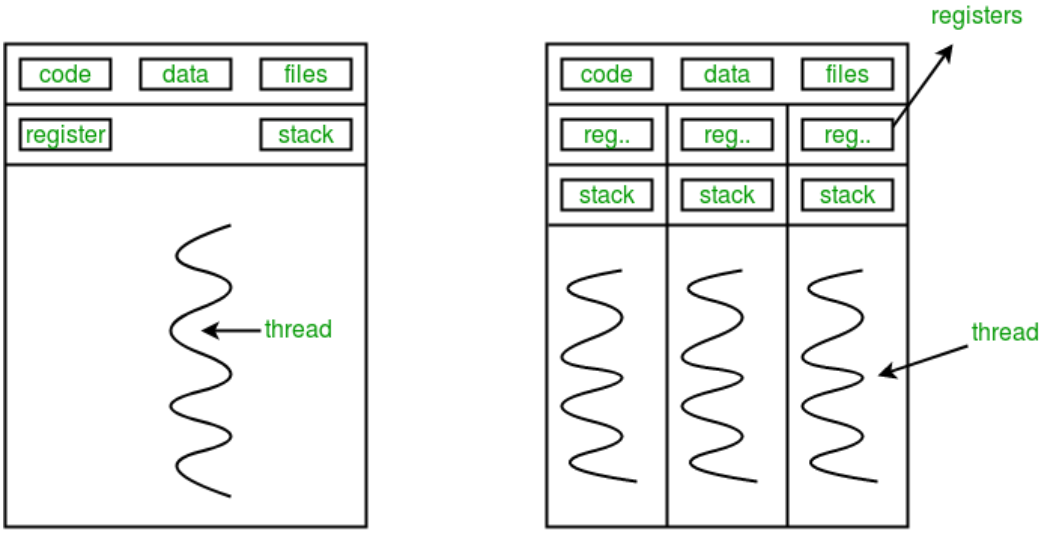
UD02: Programación multihilo



1. Fundamentos de la programación multihilo

En muchas ocasiones las sentencias que forman un programa deben ejecutarse de manera secuencial, ya que, entre todas, forman una lista ordenada de pasos a seguir para resolver un problema. Otras veces, dichos pasos no tienen por qué ser secuenciales, pudiéndose realizar varios a la vez.

Otras veces, disponer de simultaneidad en la ejecución no es opcional, sino necesario, como ocurre en las aplicaciones web. Si no se atendiesen simultáneamente las peticiones que recibe un servidor web, cada usuario debería esperar hasta que todos los usuarios que llegaron antes que él fuesen atendidos para obtener las páginas o recursos solicitados.

En ambos casos, la solución se encuentra en la programación multihilo, una técnica utilizada para conseguir procesamiento simultáneo.

Aunque plena de posibilidades, esta técnica no está exenta de condiciones y restricciones. En esta unidad se presentan las herramientas de procesamiento multihilo disponibles en el lenguaje Java, junto con los aspectos teóricos referentes al análisis de los procesos para determinar si es posible ejecutarlos con múltiples hilos, así como a detectar y prevenir los conflictos que surgen como consecuencia de la programación concurrente.

Desde el punto de vista del sistema operativo, un programa informático en ejecución es un proceso que compite con los demás procesos por acceder a los recursos del sistema. Visto desde dentro, un programa es, básicamente, una sucesión de sentencias que se ejecutan una detrás de otra.

El sistema operativo se encarga de hacer convivir los diferentes procesos y de repartir los recursos entre sí, por lo que cuando se programa no hay que tener en cuenta aspectos relacionados con cómo se van a gestionar los procesos o cómo estos van a tener acceso a los recursos. Salvo el optimizar el uso de memoria y conseguir que los algoritmos sean eficientes, los programadores no tienen que preocuparse del resto: el sistema operativo se encarga de la concurrencia a nivel de proceso.

Es dentro del interior de los procesos donde la programación tiene algo que decir con respecto a la concurrencia, mediante la programación multihilo.

> Un hilo, también conocido como thread, es una pequeña unidad de computación que se ejecuta dentro del contexto de un proceso. Todos los programas utilizan hilos.

En el caso de un programa absolutamente secuencial, el hilo de ejecución es único, lo que provoca que cada sentencia tenga que esperar que la sentencia inmediatamente anterior se ejecute completamente antes de comenzar su ejecución. Esto no quiere decir en ningún caso que no existan estructuras de control, como `if` o `while`, sino que las sentencias se van ejecutando una detrás de otra y no de manera simultánea.

En cambio, en un programa multihilo, algunas de las sentencias se ejecutan simultáneamente, ya que los hilos creados y activos en un momento dado acceden a los recursos de procesamiento sin necesitar esperar a que otras partes del programa terminen.

A continuación, se citan algunas de las características que tienen los hilos:

- **Dependencia del proceso**. No se pueden ejecutar independientemente. Siempre se tienen que ejecutar dentro del contexto de un proceso.
- **Ligereza**. Al ejecutarse dentro del contexto de un proceso, no requiere generar procesos nuevos, por lo que son óptimos desde el punto de vista del uso de recursos. Se pueden generar gran cantidad de hilos sin que provoquen pérdidas de memoria.
- **Compartición de recursos**. Dentro del mismo proceso, los hilos comparten espacio de memoria. Esto implica que pueden sufrir colisiones en los accesos a las variables provocando errores de concurrencia.
- **Paralelismo**. Aprovechan los núcleos del procesador generando un paralelismo real, siempre dentro de las capacidades del procesador.
- **Concurrencia**. Permiten atender de manera concurrente múltiples peticiones. Esto es especialmente importante en servidores web y de bases de datos, por ejemplo.

Para ilustrar de manera más precisa cómo se comporta un programa de un único hilo frente a los programas de múltiples hilos se puede establecer la siguiente analogía.

El camarero de una cafetería recibe a un cliente que le pide un café, una tostada y una tortilla francesa. Si el camarero trabaja como un proceso de un único hilo pondrá la cafetera a preparar el café, esperará a que termine para poner el pan a tostar y no pedirá la tortilla a la cocina hasta que el pan no esté tostado. Probablemente, cuando todo esté dispuesto para servir, el café y la tostada estarán fríos y el cliente aburrido de esperar.

Los camareros (la gran mayoría de ellos) suelen trabajar como procesos multihilo: ponen la cafetera a preparar el café, el pan a tostar y piden a la cocina los platos sin esperar a que cada una de las demás tareas esté terminada. Dado que cada una de ellas consume diferentes recursos, no es necesario hacerlo. Transcurrido el tiempo que tarda en estar lista la tarea más lenta, el cliente estará atendido.

Siguiendo con la analogía, al igual que pasa en los ordenadores, los recursos de una cafetería son limitados, por lo que hay una restricción física que impide que se hagan más tareas simultáneamente de las que permiten los recursos. Si solo hay una tostadora con capacidad para dos rebanadas de pan no se podrán tostar más de dicho número en un mismo instante de tiempo.

La programación que permite realizar este tipo de sistemas se llama multihilo, concurrente o asíncrona.

1.1. Programación secuencial o de único hilo

El siguiente ejemplo programado en Java ilustra cómo se comporta un programa que se ejecuta en un único hilo, así como las consecuencias que esto implica. El programa está compuesto por una única clase (representa un ratón) compuesta por dos atributos: el nombre y el tiempo en segundos que tarda en comer. En el método main se instancian varios objetos (ratones) y se invoca al método comer de cada uno de ellos. Este método muestra un texto por pantalla cuando comienza, realiza una pausa de la duración en se- gundos (con el método sleep de la clase Thread) que indica el parámetro tiempoAlimentacion y, finalmente, muestra otro texto por pantalla cuando finaliza.

Consulta el Ejemplo01

1.2. Programación concurrente o multihilo

El ejemplo anterior se puede programar de manera concurrente de forma sencilla, ya que no hay ningún recurso compartido. Para ello, basta con convertir cada objeto de la clase Raton en un hilo (thread) y programar aquello que se quiere que ocurra concurrentemente dentro del método run. Una vez creadas las instancias, se invoca al método start de cada una de ellas, lo que provoca la ejecución del contenido del método run en un hilo independiente.

Consulta el Ejemplo02

En Java existen dos formas para la creación de hilos:

- Implementando la interface `java.lang.Runnable`.
- Heredando de la clase `java.lang.Thread`.

La implementación de la interface `Runnable` obliga a programar el método sin argumentos `run`.

```
1 public class ThreadViaInterface implements Runnable {
2     @override
3     public void run() {
4     }
5 }
```

Heredando de la clase Thread esto no es obligatorio porque dicha clase ya es una implementación de la interface Runnable.

```
1 public class ThreadViaInheritance extends Thread {
2 }
```

No obstante, crear un hilo heredando de `Thread` «necesita» la implementación del método sin argumentos `run`, ya que de lo contrario la clase no tendrá un comportamiento multihilo.

Consulta el Ejemplo3

> Es habitual que, en las primeras tomas de contacto con los hilos en Java, los programadores intenten ejecutar el método `run` en lugar de ejecutar el método `start`. Desde el punto de vista de la compilación no habrá problema: el programa se compilará sin errores.
>
> **Desde la perspectiva de la programación concurrente el resultado no será el adecuado, ya que los hilos se ejecutarán uno detrás de otro, de manera secuencial, no obteniendo ninguna mejora frente a la programación secuencial convencional.**

La creación de threads mediante la implementación de la interface `Runnable` tiene una ventaja clara sobre la herencia de la clase `Thread`: al ser Java un lenguaje que no admite herencia múltiple, heredar de dicha clase impide otro tipo de herencias, limitando las posibilidades de diseño del software.

Por otra parte, a través de la implementación mediante la interface `Runnable` se pueden lanzar muchos hilos de ejecución sobre un único objeto, frente a la herencia de `Thread` que creará un objeto por cada hilo.

En el siguiente ejemplo se implementa un hilo mediante la interface `Runnable` para crear múltiples hilos a partir de un único objeto. En el hilo, un atributo de instancia llamado `alimentoConsumido` se incrementa en 1 durante la ejecución del método `comer`, invocado en el método `run`. Se puede observar en el método `main` cómo se crea una única instancia de la clase `RatonSimple` y se crean cuatro hilos que la

ejecutan.

Consulta el Ejemplo04 y Ejemplo04bis

> Debido a la naturaleza de la programación multihilo, las salidas de las ejecuciones de los programas de ejemplo pueden no coincidir con las presentadas en los apuntes. De hecho, en distintas ejecuciones en la misma máquina los resultados pueden variar.

1.3. Estados de los hilos

Durante el ciclo de vida de los hilos, estos pasan por diversos estados. En Java, están recogidos dentro de la enumeración `State` contenida dentro de la clase `java.lang.Thread`.

El estado de un hilo se obtiene mediante el método `getState()` de la clase `Thread`, que devolverá algunos de los valores posibles recogidos en la enumeración indicada anteriormente.

Los estados de un thread son los que se muestran en la siguiente tabla:

Estado	Valor en `Thread.State`	Descripción
Nuevo	`NEW`	El hilo está creado, pero aún no se ha arrancado.
Ejecutado	`RUNNABLE`	El hilo está arrancado y podría estar en ejecución o pendiente de ejecución.
Bloqueado	`BLOCKED`	Bloqueado por un monitor.
Esperando	`WAITING`	El hilo está esperando a que otro hilo realice una acción determinada.
Esperando un tiempo	`TIME_WAITING`	El hilo está esperando a que otro hilo realice una acción determinada en un período de tiempo concreto.
Finalizado	`TERMINATED`	El hilo ha terminado su ejecución.

En el siguiente código de ejemplo, se almacenan en un `ArrayList` los estados por los que pasa un hilo que contiene en su interior una llamada al método `sleep`. Se utiliza la clase `RatonSimple` que implementa `Runnable` de los ejemplos anteriores.

Consulta el Ejemplo05

> Todos los hilos pasan por los estados **NEW**, **RUNNABLE** y **TERMINATED**. El resto de los estados están condicionados a circunstancias propias de la ejecución.

1.4. Introducción a los problemas de concurrencia

En programación concurrente las dificultades aparecen cuando los distintos hilos acceden a un recurso compartido y limitado. Si en el ejemplo de los ratones (el físico, no el programático), estos comiesen a través de un dispositivo al que solo pudiesen acceder de uno en uno, la concurrencia sería imposible. En el ejemplo programático el problema es el mismo, pero no existe la restricción física, por lo que nada impide

que los hilos accedan al recurso compartido y es en ese momento donde aparecerán los problemas.

El acceso a los recursos compartidos limitados, por lo tanto, se debe gestionar correctamente. Por suerte, existen técnicas, clases y librerías que implementan soluciones, por lo que únicamente hay que conocerlas y saberlas utilizar en el momento adecuado.

Un aspecto importante referente a los problemas de concurrencia es que no siempre provocan un error en tiempo de ejecución. Esto significa que en sucesivas ejecuciones del programa multihilo el error se producirá en algunas de ellas y en otras no. Frente al determinismo de los programas secuenciales (los mismos datos de entrada generan siempre las mismas salidas), en el entorno multihilo los factores que determinan las condiciones de ejecución pueden tener su origen en elementos sobre los que el programador no tiene capacidad de influir, como el sistema operativo.

La dificultad, en este tipo de programación, reside en el hecho de que el programador tiene que saber de forma anticipada que el error se puede producir en una sentencia o bloque de sentencias, por cómo esta se ejecuta y qué recursos utiliza. No basta simplemente con realizar pruebas convencionales para determinar que el algoritmo está bien programado. Además, hay que «saber» que está bien programado.

Como se ha indicado anteriormente existen soluciones para todos los problemas, pero en esta afirmación hay una verdad a medias. Los problemas de la programación concurrente se resuelven, en ciertos casos, convirtiendo en secuenciales determinadas partes del código. Si no se programa correctamente, se corre el riesgo de convertir todo el programa en secuencial lo que evitaría, efectivamente, los problemas de concurrencia ya que esta habría dejado de existir.

2. Programación multihilo: clases y librerías

2.1. Paquete `java.lang`

Java dispone de un importante número de clases destinadas a la programación multihilo. Estas clases tienen múltiples aplicaciones, desde la propia construcción de los hilos hasta dar soporte a estructuras de datos consistentes cuando varios hilos acceden a ellas.

Estas clases se encuentran agrupadas en dos paquetes principales: el paquete `java.lang` y el paquete `java.util.concurrent`.

El paquete `java.lang` es un paquete importado por defecto en Java (contiene las clases básicas, incluida `Object` como clase raíz de la jerarquía de clases del lenguaje) por lo que no hay que importar ninguna librería para utilizar sus clases.

Dentro de este paquete se encuentran la interfaz `Runnable` y la clase `Thread`, como elementos fundamentales para la construcción de hilos. También se encuentran las clases `Timer` y `TimerTask`

Nombre	Tipo	Descripción
`Runnable`	Interface	Esta interface debe implementarse por aquellas clases que se quieran ejecutar como un `thread`. Define un método sin argumentos `run`.
`Thread`	Clase	Esta clase implementa la interface `Runnable`, siendo un hilo en sí misma. Una clase que hereda de `Thread` y que sobrescribe el método `run` se ejecuta de manera concurrente si se invoca al método `start`. Con la clase `Thread` se puede envolver un objeto que implemente `Runnable` provocando que se ejecute en un hilo independiente.
`Timer`	Clase	Permite la programación de la ejecución de tareas en diferido y de forma repetitiva mediante el método `schedule`. Este método espera tareas programables, representadas por objetos de la clase `TimerTask`, así como información del momento de inicio de la tarea y del tiempo que debe transcurrir entre cada una de las ejecuciones de la misma.
`TimerTask`	Clase abstracta	Heredando de esta clase y sobrescribiendo el método `run` se puede crear una tarea programable con la clase `Timer`.

Consulta el Ejemplo06

2.2. Paquete `java.util.concurrent`

En este paquete se incluyen un interesante conjunto de clases y herramientas relacionadas con la programación concurrente. A continuación, se muestran algunos de los elementos más relevantes del mismo.

2.2.1. Executor

Es una interface para la definición de sistemas multihilo. Permite ejecutar tareas de tipo `Runnable`. Algunas de sus interfaces derivadas, así como clases directamente relaciona das se muestran en la siguiente tabla.

Nombre	Tipo	Descripción
`ExecutorService`	Interface	Subinterface de Executor, permite gestionar tareas asíncronas
`SchedulledExecutorService`	Interface	Permite la planificación de la ejecución de tareas asíncronas.
`Executors`	Clase	Fábrica de objetos `Executor`, `ExecutorServices`, `ThreadFactory` y `Callable`.
`TimeUnit`	Enumeración	Proporciona representaciones de unidades de tiempo con distinta granularidad, desde días hasta nanosegundos.

Consulta el Ejemplo07

2.3. Queues

Java proporciona componentes que resuelven todo el espectro de necesidades relacionado con las colas para crear entornos seguros de ejecución de soluciones de mensajería, gestión de colas de trabajo y sistemas basados en esquemas productor-consumidor en entornos concurrentes.

Las clases más relevantes de este grupo se recogen en la siguiente tabla.

Nombre	Tipo	Descripción
`ConcurrentLinkedQueue`	Clase	Una cola enlazada resistente a hilos.
`ConcurrentLinkedDeque`	Clase	Una cola doblemente enlazada resistente a hilos.
`BlockingQueue`	Interface	Una cola que incluye bloqueos de espera para la gestión del espacio. Algunas de sus implementaciones son `LinkedBlockingQueue`, `ArrayBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue` y `DelayQueue`.
`TransferQueue`	Interface	Un tipo de `BlockingQueue` especialmente diseñado para mensajería.
`BlockingDeque`	Interface	Una cola doblemente enlazada que incluye bloqueos de espera para la gestión del espacio. Dispone de una implementación en la clase `LinkedBlockingDeque`.

Consulta el Ejemplo08 y Ejemplo08bis

2.4. Sincronizadores

El paquete `java.util.concurrent` proporciona cinco clases específicas para conseguir que la concurrencia entre hilos se desarrolle de manera correcta. Estas clases son las que se recogen en la siguiente tabla.

Nombre	Tipo	Descripción
`Semaphore`	Clase	Proporciona el mecanismo clásico para regular el acceso de los a recursos de uso limitado.
`CountDownLatch`	Clase	Proporciona una ayuda para la sincronización cuando los hilos deben esperar hasta que se realicen un conjunto de operaciones.
`CyclingBarrier`	Clase	Proporciona una ayuda para la sincronización cuando unos hilos deben esperar hasta que otros hilos alcancen un punto de ejecución determinado.
`Phaser`	Clase	Proporcionauna funcionalidad similar a `CountDownLatch` y a `CyclingBarrier` a través de un uso más flexible.
`Exchanger`	Clase	Permite intercambiar elementos entre dos hilos.

Consulta el Ejemplo09

2.5. Estructuras de datos concurrentes

Java dispone de interfaces y clases para almacenar información con prácticamente cualquier tipo de estructura de datos. Interfaces heredadas de la interface `Collection`, como `List`, `Map`, `Set`, `Qeue` o `Deque`, son la base de una serie de clases que, implementando dichas interfaces, proporcionan el soporte necesario para todo tipo de estructuras de datos.

Desde el punto de vista de la concurrencia, estas implementaciones no están diseñadas para soportar que múltiples hilos lean y escriban simultáneamente en ellas, pudiendo provocar errores.

En el Ejemplo10, varios hilos leen y escriben en un objeto `ArrayList` compartido.

La ejecución de este código provoca múltiples errores de concurrencia.

La clase `Collections` de Java proporciona métodos estáticos para convertir estructuras de datos en seguras respecto a hilos (thread-safe), como `synchronizedList`, `synchronizedMap` o `synchronizedSet` entre otros, pero no son las alternativas más eficientes en todos los casos.

Como alternativa más eficiente si la mayor parte de las operaciones son de lectura, el paquete `java.util.concurrent` proporciona implementaciones específicamente diseñadas para entornos de ejecución multihilo. Algunas de esas clases se muestran en la siguiente tabla.

Nombre	Tipo	Descripción
`ConcurrentHashMap`	Clase	Equivalente a un `HashMap` sincronizado.
`ConcurrentSkipListMap`	Clase	Equivalente a un `TreeMap` sincronizado.
`CopyOnWriteArrayList`	Clase	Equivalente a un `ArrayList` sincronizado.
`CopyOnWriteArraySet`	Clase	Equivalente a un `Set` sincronizado.

El ejemplo anterior, referente a los lectores y escritores trabajando conjuntamente sobre un `ArrayList`, se convierte en thead-safe utilizando la clase `CopyOnWriteArraySet` en lugar de `ArrayList`, tal y como se puede observar en el Ejemplo11.

2.6. Las interfaces `ExecutorService`, `Callable` y `Future`

Usadas de manera conjunta, estas tres interfaces proporcionan mecanismos para ejecutar el código de manera asíncrona.

`ExecutorService` proporciona el marco de ejecución de código asíncrono contenido en objetos de las interfaces `Callable` y `Runnable`. Sus principales métodos se muestran en la siguiente tabla.

Método	Descripción
`awaitTermination`	Bloquea el servicio cuando se recibe una petición de apagado hasta que todas las tareas que tenía asignadas han terminado o se ha alcanzado el tiempo límite E de espera (timeout) o ha habido una interrupción.
`invokeAll`	Permite lanzar una colección de tareas y recoger una lista de objetos `Future`.
`invokeAny`	Permite lanzar una colección de tareas y recoger el resultado de la que termine satisfactoriamente (si alguna lo hace).
`shutdown()`	Hace que que están el servicio en ejecución deje de y finaliza. aceptar nuevas tareas, espera a que terminen
`shutdownNow()`	Finaliza el servicio sin esperar a que las tareas que tiene en ejecución lo hagan.
`submit()`	Envía a ejecución una tarea `Runnable` o `Callable`.

La construcción de instancias de `ExecutorService` se realiza a través de una serie de métodos estáticos de la clase Executors. Estos métodos permiten indicar la estrategia de hilos que desea que siga el ExecutorService. Algunos de los métodos estáticos de Executors para crear objetos de la interfaz ExecutorService son:

- `Executors.newCachedThreadPool()`: Crea un `ExecutorService` con un pool de hilos con todos los que sean necesarios, reutilizando aquellos que están libres.
- `Executors.newFixedThreadPool(int nThreads)`: Crea un `ExecutorService` con un pool con un número determinado de hilos.
- `Executors.newSingleThreadExecutor()`: Crea un `ExecutorService` con un único hilo.

Por su parte, la interfaz `Callable` es una interfaz que funciona similar a `Runnable`, pero con la diferencia de que puede devolver un retorno y lanzar una excepción. Es una interfaz funcional, ya que solo tiene el método call, por lo que se puede utilizar en expresiones lambda. El código asíncrono de un objeto `Callable` se ejecuta a través del método submit de `ExecutorService`.

La interfaz `Future` representa un resultado futuro generado por un proceso asíncrono. De alguna manera, es un sistema que permite dejar en suspenso la obtención de un resultado hasta que este está disponible. El resultado de invocar al método `submit` de un `ExecutorService` es un objeto `Future`.

Los métodos de la interfaz `Future` se recogen en la siguiente tabla.

Método	Descripción
`cancel`	Intenta cancelar la ejecución de la tarea.
`get`	Espera a que para termine la tarea y obtiene el resultado. En una de sus formas admite un timeout para limitar el tiempo de espera.
`isCancelled()`	Indica si la tarea se ha cancelado antes de terminar.
`isDone()`	Indica si la tarea ha terminado.

En resumen, la relación entre estas tres interfaces es la siguiente: `ExecutorService` ejecuta un objeto `Callable` (o `Runnable`) con una estrategia multihilo determinada y deposita en un objeto `Future` el retorno futuro de la tarea.

Consulta el Ejemplo12

> En programación, casi nunca una alternativa es siempre mejor que otra. Las interfaces `Runnable` y `Callable` son absolutamente válidas y, dependiendo de la situación, una u otra serán la mejor alternativa.

3. Programación asíncrona

Conceptualmente, la programación asíncrona es independiente del lenguaje de programación en el que se realice, siempre y cuando este la admita. En cambio, a nivel práctico, cada lenguaje de programación proporciona distintas herramientas para permitir la creación y gestión de hilos.

Java es un lenguaje con una extensa biblioteca de clases orientadas a la programación asíncrona o multihilo. Gracias a muchas de estas clases se pueden resolver problemas extremadamente complejos sin excesiva dificultad, ya que implementan soluciones que, de no existir, habría que desarrollar.

En este apartado se muestran los mecanismos básicos de creación, ejecución y sincronización de hilos. Aunque algunos de estos mecanismos ya han sido previamente presentados en esta unidad, se vuelven a presentar de manera resumida con el fin de ofrecer una visión panorámica de los elementos básicos de la programación asíncrona en Java.

3.1. La interfaz `Runnable`

Una clase que implementa la interfaz `Runnable` está concebida para ejecutarse dentro de un thread. El método `run` es su único método y este no tiene argumentos ni retorno. El contenido del método `run` se ejecuta de manera asíncrona, por lo que el hilo principal de la aplicación no se detiene, aunque el bloque de código contenido en dicho método lo haga.

Cuando se instancia un objeto de una clase que implementa `Runnable`, esta se tiene que ejecutar a partir del método `start` de la instancia de `Thread` construida a partir del objeto `Runnable`.

En el Ejemplo13 se muestra la creación y ejecución de 10 threads utilizando objetos de la interfaz `Runnable`.

Al tratarse de una interfaz funcional, `Runnable` se puede utilizar en una expresión lambda, disponible en Java desde la versión 8.

Consulta el Ejemplo14 donde se muestra una versión modificada del Ejemplo13 usando la expresión lambda.

3.2. La clase `Thread`

La clase `Thread` representa un hilo de ejecución. Cuando una clase hereda de `Thread` puede implementar el método `run` y ejecutarse de forma asíncrona.

Para lanzar un objeto `Thread` de manera asíncrona, basta ejecutar su método `start`. El Ejemplo15 muestra la creación y ejecución de un hilo básico utilizando la clase `Thread`.

En Java, los hilos normales se denominan hilos de usuario, existiendo otro tipo conocido como hilos demonios. Estos hilos se crean a partir de un hilo convencional mediante el método `setDaemon`. Un hilo demonio constituye un subproceso con una prioridad inferior a la de un hilo de usuario, ejecutándose después. La otra diferencia es que este tipo de hilos tienen utilidad cuando existen hilos de usuario, por lo que cuando estos últimos terminan, finalizan los hilos daemon.

Los métodos principales de la clase `Thread` se muestran en la siguiente tabla.

Método	Descripción
`start`	Método de inicio del bloque asíncrono de la clase.
`run`	Método en el que se programa el bloque asíncrono. Se ejecuta cuando se invoca el método start.
`join`	Bloquea el hilo hasta que termina el hilo referenciado.
`sleep`	Método estático que detiene temporalmente la ejecución del hilo.
`getId`	Devuelve el identificador del hilo. Es un long positivo generado cuando se crea el hilo.
`getName`	Devuelve el nombre del hilo, asignado en algunas de las formas del constructor.
`getState`	Devuelve el estado del hilo como un valor de la enumeración `Thread.State`.
`interrupt`	Interrumpe la ejecución del hilo.
`interrupted`	Método estático que comprueba si el hilo actual ha sido interrumpido.
`isInterrupted`	Comprueba si el hilo en el que se encuentra ha sido interrumpido.
`isAlive`	Comprueba si el hilo está vivo.
`setPriority`	Cambia la prioridad del hilo. El valor debe estar entre las constantes `MIN_PRIORITY` y `MAX_PRIORITY` de la propia clase `Thread`. El uso que se haga de las prioridades establecidas viene determinado por el sistema operativo.
`setDaemon`	Permite marcar el hilo como un hilo demonio.
`isDaemon`	Determina si un hilo es un hilo demonio.
`yield`	Método estático que indica al planificador que está dispuesto a ceder su uso actual de procesador. El planificador decide si atiende o no esta sugerencia.

3.3. Suspensión de ejecución: el método `sleep`

El método estático `sleep` de la clase `Thread` permite suspender la ejecución del hilo desde el que se invoca. Acepta como parámetro la cantidad de tiempo en milisegundos que se desea realizar esta suspensión, cuya precisión queda sometida a la precisión de los temporizadores del sistema y al planificador.

3.4. Interrupciones

En computación, una interrupción es una suspensión temporal de la ejecución de un proceso o de un hilo de ejecución. Las interrupciones no suelen pertenece a los programas, sino al sistema operativo, viniendo generadas por peticiones realizadas por los dispositivos periféricos.

En Java, una interrupción es una indicación a un hilo de que debe detener su ejecución para hacer otra cosa. Es responsabilidad del programador decidir qué quiere hacer ante una interrupción, siendo lo más habitual detener la ejecución del hilo.

Las interrupciones se capturan mediante la excepción InterruptedException, provocada por algunas operaciones como la invocación al método sleep. En caso de que la excepción no se tenga que controlar, se deberá invocar al método estático interrupted para Paraninto gestionar la interrupción.

En el Ejemplo16 se realiza la gestión de una excepción que se lanza de manera forzada ante cierta condición. En la captura de la excepción se invoca a la sentencia `return` para finalizar la ejecución del hilo.

3.5. Compartición de información

Varios hilos pueden crearse como instancias de la clase Thread. Los atributos de dichos hilos, si no son estáticos, serán específicos de cada uno de ellos, por lo que no se podrán utilizar para compartir información.

Si se desea que varios hilos compartan información existen varias alternativas:

- **Utilizar atributos estáticos**. Los atributos estáticos son comunes a todas las instancias, por los que independientemente de la manera de construir los hilos la información es compartida.
- **Utilizando referencias a objetos comunes** accesibles desde todos los hilos. En el Ejemplo17 se muestra un ejemplo de uso de una instancia de un objeto común a varios hilos que se modifican en cada uno de ellos.
- **Utilizando atributos no estáticos** de la instancia de una clase que implemente `RunnableSharing` y construyendo los hilos a partir de dicha instancia. El Ejemplo18 muestra un ejemplo.

Existen otras formas de compartir información, ya sea a través de ficheros, bases de datos o servicios de red o de internet. En todos los casos hay que tener en cuenta que la información compartida por varios hilos para lectura y escritura es una potencial fuente de errores de concurrencia.

4. Problemas y soluciones de la programación concurrente. Sincronización

De manera intrínseca, la programación concurrente tiene dos características que pueden ser fuentes de errores: los recursos compartidos y el orden de ejecución.

En lo referente a la compartición de recursos la problemática es diversa, ya que puede afectar a cómo se modifica el valor de una variable, se accede a los datos de una estructura de datos, se ejecuta un bloque de código o se accede a un recurso limitado.

Para ilustrar este tipo de problemas se puede utilizar el Ejemplo19.

Es representativo de la problemática de la programación concurrente que en una operación aparentemente tan sencilla puedan surgir errores tan importantes.

> Los problemas de concurrencia no se producen siempre de la misma manera, ya que la ejecución no es determinista. El algoritmo que genera un error cuando 1000 hilos incrementan en 1000 unidades una variable compartida puede funcionar correctamente en la mayoría de los casos con 10 hilos y un incremento de 10 unidades por hilo. No obstante, el riesgo de que un algoritmo pueda fallar, aunque la posibilidad sea baja, normalmente es inaceptable en computación.

El orden de ejecución, por su parte, tiene que ver con las dependencias que se pueden producir entre los bloques de código en función del orden de ejecución. Si, por ejemplo, un bloque de código necesita como entrada datos generados como salida por otro bloque, esto provocará una dependencia, dado que en un sistema multihilo no se tiene control sobre el orden de las ejecuciones salvo que se establezcan mecanismos de sincronización.

En este apartado se exponen los términos y conceptos más importantes relacionados con los problemas de concurrencia, así como dichos problemas y sus posibles soluciones.

4.1. Conceptos

Para poder comprender tanto los problemas como las soluciones relacionados con la programación concurrente es necesario conocer el significado de algunos conceptos. En este apartado se exponen los más importantes.

4.1.1. Recursos compartidos

Un recurso compartido es, como su propio nombre indica, un elemento del sistema que es utilizado por varios hilos de ejecución simultáneamente. Puede ser un atributo estático de una clase o un atributo no estático de un objeto compartido por todos los hilos, un método estático de una clase o un método no estático de un objeto compartido por todos los hilos, una conexión a una base de datos, un socket o cualquier elemento con un número limitado de instancias.

4.1.2. Dependencias

Como ya se ha comentado anteriormente, no todas las tareas se pueden ejecutar en un entorno multihilo. Para poder hacerlo hay que tener la certeza de que los segmentos de código que se van a ejecutar en paralelo son independientes y el orden en el que se ejecutan es irrelevante.

Una tarea no se podrá ejecutar en un entorno multihilo si tiene dependencias. Existen tres tipos de dependencias principales:

- Dependencias de datos. Varios segmentos de código utilizan el mismo dato.
- Dependencias de flujos. Debido a que el orden de ejecución del programa no se puede determinar por existir instrucciones de control de flujo existe dependencias potenciales que no se pueden determinar en un análisis estático del código.
- Dependencias de recursos. Varios segmentos de código acceden simultáneamente a recursos del procesador.

La existencia de dependencias de cualquier tipo impide la programación concurrente salvo que se puedan establecer mecanismos de sincronización.

4.1.3. Condiciones de Bernstein

De manera formal, el cumplimiento de las condiciones de **Bernstein** determina si dos segmentos de código pueden ser ejecutados en paralelo. Dados dos segmentos de código S_1, y S_2 se determina que son independientes y pueden ser ejecutados en paralelo si:

- Las entradas de S_2 son distintas de las salidas de S_1. De no ser así se produce lo que se conoce como **dependencia de flujo**.
- Las entradas de S_1 son distintas de las salidas de S_2. De no ser así se produce lo que se conoce como **antidependencia**.
- Las salidas de S_1 son distintas de las salidas de S_2. De no ser así se produce lo que se conoce como **dependencia de salida**.

Si dos segmentos de código cumplen con las condiciones de **Bernstein** su ejecución se puede paralelizar.

4.1.4. Acción y acceso atómicos

Una acción atómica es aquella que se ejecuta sin interrupciones, de una única vez. Cualquier efecto de la acción solo es visible al finalizar la misma.

En Java, algunas acciones sencillas son atómicas:

- Leer y escribir variables de los tipos primitivos, excepto los tipos `long` y `double`.
- Leer y escribir todas las variables declaradas `volatile`, incluidos los tipos `long` y `double`.

> Acciones muy sencillas, como incrementar en 1 el valor de una variable de tipo `int` no son atómicas, por lo que hay que evaluar si es necesario establecer un mecanismo de sincronización.

4.1.5. Sección crítica

La sección crítica de un programa multihilo es el bloque de código que accede a recursos compartidos, por lo la que solo debe ser accedido por un único hilo de ejecución. Determinar correctamente la sección crítica permite sincronizar correctamente el programa para evitar errores de concurrencia, así como hacerlo eficiente para poder aprovechar al Ediciones máximo el paralelismo. El garantizar que a la sección crítica solo acceda un hilo de ejecución es lo que se conoce como **exclusión mutua**.

4.1.6. Exclusión mutua

Se denomina así a la técnica de programación consistente en hacer que, en un entorno concurrente, un proceso excluya a todos los demás del uso de un recurso compartido (una sección crítica) para garantizar la integridad del sistema.

4.1.7. Seguridad en hilos, seguro frente a hilos o `Thread safety`

Estos términos hacen referencia a la propiedad que tiene un elemento de software (una clase o una estructura de datos, por ejemplo) para ser ejecutado en un entorno de múltiples hilos de forma segura.

> En Java, la documentación de las estructuras de datos suele especificar si son `Thread safety` o en cambio no están sincronizadas (no son seguras frente a hilos). Es conveniente revisar la documentación de las clases que se utilicen por primera vez en general, prestando especial atención a las referencias a los threads en entornos multihilo.

4.2. Problemas de la programación concurrente

En esta sección se presentan algunos de los problemas que surgen como consecuencia de la programación multihilo, así como las soluciones que se pueden adoptar, tanto genéricas como específicas del lenguaje Java.

4.2.1. Interbloqueo o deadlock

Se produce cuando dos o más hilos están bloqueados entre sí. Por ejemplo, si el hilo A está esperando a que termine el hilo B para continuar su proceso y este, a su vez, está esperando a que termine el hilo A para continuar su proceso se produce un interbloqueo.

4.2.2. Muerte por inanición

Este problema se produce cuando se establece una política de prioridades que provoca que algunos hilos nunca tengan acceso a la CPU.

4.2.3. Condiciones de carrera

Se produce cuando dos bloques de código concurrente tienen dependencias entre los datos entrada y salida y no se ejecutan en el orden correcto (dependencia de flujo o antidependencia).

4.2.4. Inconsistencia de memoria

Ocurre cuando dos o más hilos tienen simultáneamente valores diferentes para la misma variable.

4.2.5. Condiciones deslizadas

Se produce cuando en un proceso se evalúa una condición para determinar si se tiene que ejecutar una sección de código y, después de la evaluación y antes de la ejecución, la condición cambia de valor. Se estaría ejecutando un bloque de código cuya condición no se está cumpliendo.

4.3. Sincronización básica: variables `volatile`

En un entorno de computación de múltiples núcleos, los procesadores disponen de técnicas de optimización y algunas de ellas se basan en el uso de memoria caché. Estas técnicas habitualmente son ventajosas, pero en la programación concurrente pueden ser una fuente de errores.

Cuando varios hilos comparten la misma variable, si esta se almacena en las cachés de los núcleos, puede ser que los hilos vean copias distintas de la misma variable, lo que puede provocar inconsistencia de memoria.

Para evitar que una variable se almacene en la caché de procesador y que todos los hilos accedan a la misma copia en Java se utiliza la palabra clave volatile. Declarando una variable como volatile solo existirá una copia en el procesador.

El siguiente código de ejemplo muestra la declaración de una variable volatile.

```
1 private volatile static long contador
```

Esta solución no resuelve por sí sola todos los problemas de inconsistencia de memoria. Si varios hilos modifican concurrentemente la misma variable, aunque sea declarada volatile, se podría seguir produciendo (habría que incluir mecanismos de sincronización).

Las variables `volatile` serían apropiadas para sistemas en los que un único hilo modifica el valor de la variable y el resto solamente lo consultan.

4.4. Sincronización básica: `wait`, `notify` y `notifyAll`

Los métodos `wait`, `notity` y `notifyAll` son propios de la clase `Object`, por lo que todas las clases en Java disponen de ellos.

Todos estos métodos se tienen que invocar desde segmentos de código de un hilo que disponga de un monitor, como, por ejemplo, un bloque o un segmento sincronizados, y obliga a capturar una excepción del tipo `InterruptedException`.

El método `wait` detiene la ejecución del hilo y los métodos `notify` y `notifiyAll` producen la reactivación de los hilos detenidos. El método `notify` hace continuar a un único segmento al azar de los que están detenidos con `wait` mientras que `notifyAll` hace continuar a todos los segmentos detenidos con `wait`.

En el Ejemplo20 se muestra un ejemplo de uso de estos métodos de sincronización. En él, dos hilos creados a partir del mismo objeto ejecutan dos métodos distintos. El primero de los hilos, al realizar la mitad de la tarea entra en espera hasta que el segundo de los hilos finaliza y notifica que debe reanudar la ejecución.

4.5. Sincronización básica: el método `join`

El método `join` permite indicar a un hilo que debe suspender su ejecución hasta que termina otro hilo de referencia. Este método debe ejecutarse dentro del bloque asíncrono del código, ya que lo contrario no tendrá ningún efecto.

El Ejemplo21 permite ilustrar el funcionamiento de este método. En él, dos hilos ejecutan un bucle con 3 iteraciones que en ausencia de mecanismos de sincronización generarían una salida similar a esta:

```
Thread 1. Interaction 0
Thread 2. Interaction 0
Thread 1. Interaction 1
Thread 2. Interaction 1
Thread 2. Interaction 2
Thread 1. Interaction 2
```

Dado que cada hilo tiene la misma prioridad y el mismo código, escribirán simultáneamente la salida programada.

En cambio, en el Ejemplo22, después de arrancar los dos hilos, se le indica al hilo `hilo2` que debe quedar suspendido hasta que termine la ejecución de `hilo1`. Para ello es necesario que `hilo2` tenga una referencia a `hilo1` (`referenceThread` en el ejemplo) para invocar al método `join`.

El resultado obtenido en este caso es el siguiente:

```
Thread 1. Interaction 0
Thread 1. Interaction 1
Thread 1. Interaction 2
Thread 2. Interaction 0
Thread 2. Interaction 1
Thread 2. Interaction 2
```

4.6. Sincronización básica: estructuras de datos resistentes a hilos

Las estructuras de datos convencionales proporcionadas por Java satisfacen cualquier necesidad relacionada con el almacenamiento de datos en memoria. Desde el `punto de vista de la programación concurrente hay que tener en cuenta que las clases ArrayList, Vector`, `HashMap` o `HashSet`, todas ellas del paquete `java.util` no están sincronizadas, lo que implica que no están programadas para ser consistentes frente al acceso desde múltiples hilos.

Para poder utilizar estas estructuras hay que usar las técnicas de sincronización tratadas en esta unidad, convertirlas a estructuras sincronizadas con los métodos estáticos proporcionados por la clase `java.util.Collections` (por ejemplo `synchronizedList` para objetos que implementen la interfaz `List`) o utilizar las estructuras de datos proporcionadas por el paquete `java.util.concurrent` y presentadas en un apartado anterior.

4.7. Sincronización avanzada: exclusión mutua, `synchronized` y monitores

Uno de los mecanismos proporcionados por Java para sincronizar segmentos de código consiste en utilizar la palabra clave `synchronized`. Mediante el uso de synchronized se puede limitar el acceso a un segmento del código a un único hilo concurrentemente, lográndose así la exclusión mutua o mutex. Permite sincronizar tanto métodos como segmentos de código (declaraciones sincronizadas), permitiendo esta última alternativa delimitar la sección crítica con más precisión.

A nivel de método, el uso es tan sencillo como incluir la palabra en la declaración del método:

```
1 public synchronized void calculate ()
```

El efecto de esta declaración es que, para una instancia de objeto, solo un hilo puede estar ejecutando el método sincronizado en un momento dado.

Por su parte, a nivel de segmento de código, la sincronización se efectúa delimitando un bloque de sentencias haciendo una declaración sincronizada. El siguiente código muestra una declaración sincronizada.

```
1 public void calculate(){
2     //Sentences not synchronized
3     synchronized (objetoBloqueo) {
4         //Block of unsyncrhonized sentences
5     }
6     //Sentences not synchronized
7 }
```

Este sistema utiliza un concepto conocido como bloqueo intrínseco, bloqueo de `monitor` o, simplemente, `monitor`. El `monitor` es un elemento asociado a una instancia que hace la función de candado. Cuando se ejecuta un método o bloque sincronizado utilizando un determinado `monitor`, este se bloquea y no puede utilizarse hasta que no queda liberado, impidiendo ejecutar un código que utilice el mismo bloqueo.

Cuando se utilizan métodos sincronizados se usa como `monitor` el objeto en el que están. Esto supone que cuando un método sincronizado de un objeto se está ejecutando ningún otro método sincronizado de ese objeto se puede ejecutar. La exclusión es, por lo tanto, muy genérica y puede que sea poco eficiente en muchos casos.

Consulta el Ejemplo23

Como se ha indicado anteriormente, el comportamiento de los métodos sincronizados del ejemplo se debe a que se ejecutan sobre la misma instancia del objeto que implementa `Runnable`, por lo que utilizan dicho objeto como monitor provocando el bloqueo.

Si en cambio utilizasen objetos distintos, el resultado sería el mismo que al no marcar los métodos como `synchronized`, ya que utilizan bloqueos diferentes.

Consulta el Ejemplo24

Por su parte, la sincronización a nivel de segmento necesita también un `monitor`, pero al no depender del objeto en el que se está ejecutando es más flexible. Utilizando bloques sincronizados no es necesario bloquear todos los segmentos de un objeto como ocurre con los métodos de objeto, sino que se pueden agrupar en monitores distintos.

En el Ejemplo25 se realiza la sincronización a nivel de bloque, utilizando dos bloqueos distintos en cada uno de ellos. De tal manera que los métodos no son exclusivos entre sí. La sincronización se realiza a nivel de método (cada uno de los métodos solo puede estar siendo ejecutado por un objeto simultáneamente, pero ambos métodos se pueden estar ejecutando por dos objetos distintos).

En cambio, si los métodos utilizan el mismo objeto como bloqueo, cuando un objeto esté ejecutando uno de los métodos ningún objeto puede ejecutar ninguno de los dos métodos.

4.8. Sincronización avanzada: semáforos

El uso de la palabra clave `synchronized` permite establecer secciones críticas a las que únicamente tiene acceso un hilo en un momento determinado, pero existen otros escenarios en los que los recursos limitados permiten el acceso de más de un hilo.

Cuando una sección crítica permite ser ejecutada por más de un hilo, pero el número de estos está limitado se utiliza un elemento de programación concurrente conocido como semáforo e implementado en Java por la clase `Semaphore`.

Los semáforos se suelen utilizar cuando un recurso tiene una capacidad limitada y se desea controlar el número de consumidores de dicho recurso. Es, por lo tanto, un elemento de sincronización.

Al construir el semáforo se le proporciona a través del constructor una capacidad que hace referencia al número de hilos que puede estar ejecutando concurrentemente.

Esta capacidad se convierte en número de permisos de acceso que se conceden a los bloques de código que quieran hacer uso del recurso limitado. Si todos los permisos están concedidos, los hilos quedan en espera a que los propietarios de los permisos los liberen.

Los principales métodos de la clase semáforo son los que se recogen a continuación en la siguiente tabla.

Método	Descripción
`acquire`	Adquiere uno o más permisos del semáforo si hay alguno disponible. En caso contrario el hilo queda a la espera.
`release`	Libera uno o más permisos concedidos previamente.
`tryAcquire`	Intenta obtener uno o más permisos, pudiendo quedar en espera durante un tiempo limitado.

Además de estos métodos la clase `Semaphore` proporciona métodos de gestión y consulta del estado del semáforo.

En el Ejemplo26 ejemplo, se construye con semáforo con capacidad para tres permisos. A su vez, cinco hilos construidos a partir de la misma instancia de `Runnable` acceden a la sección crítica, solicitan el permiso, realizan los pasos de la actividad sincronizada y liberan el bloqueo.

> No solo se pueden utilizar los semáforos cuando los recursos son limitados. En ocasiones es conveniente limitar el número de hilos que realizan determinadas tareas para no saturar el sistema o alguno de sus componentes. Por ejemplo, limitar el número de hilos que acceden a servicios web o a bases de datos puede ser una estrategia adecuada en muchos casos.

5. Ejemplos

5.1. Ejemplo01

Ejemplo con un único hilo de ejecución:

```
1  public class Mouse {
2
3      private String name;
4      private int feedingTime;
5
6      public Mouse(String name, int feedingTime) {
7          super();
8          this.name = name;
9          this.feedingTime = feedingTime;
10     }
11
12     public void eat() {
13         try {
14             System.out.printf("The mouse %s has started to feed%n", name);
15             Thread.sleep(feedingTime * 1000);
16             System.out.printf("The mouse %s has stopped to feed%n", name);
17         } catch (InterruptedException e) {
18             e.printStackTrace();
19         }
20     }
21
22     public static void main(String[] args) {
23         Mouse fievel = new Mouse("Fievel", 4);
24         Mouse jerry = new Mouse("Jerry", 5);
25         Mouse pinky = new Mouse("Pinky", 3);
26         Mouse mickey = new Mouse("Mickey", 6);
27         fievel.eat();
28         jerry.eat();
29         pinky.eat();
30         mickey.eat();
31     }
32 }
```

La salida producida por la ejecución es la siguiente:

```
The mouse Fievel has started to feed
The mouse Fievel has stopped to feed
The mouse Jerry has started to feed
The mouse Jerry has stopped to feed
The mouse Pinky has started to feed
The mouse Pinky has stopped to feed
The mouse Mickey has started to feed
The mouse Mickey has stopped to feed
```

5.2. Ejemplo02

Ejemplo multihilo:

```
public class Mouse extends Thread {

    private String name;
    private int feedingTime;

    public Mouse(String name, int feedingTime) {
        super();
        this.name = name;
        this.feedingTime = feedingTime;
    }

    public void eat() {
        try {
            System.out.printf("The mouse %s has started to feed%n", name);
            Thread.sleep(feedingTime * 1000);
            System.out.printf("The mouse %s has stopped to feed%n", name);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void run() {
        this.eat();
    }

    public static void main(String[] args) {
        Mouse fievel = new Mouse("Fievel", 4);
        Mouse jerry = new Mouse("Jerry", 5);
        Mouse pinky = new Mouse("Pinky", 3);
        Mouse mickey = new Mouse("Mickey", 6);
        fievel.start();
        jerry.start();
        pinky.start();
        mickey.start();
    }
}
```

Ejemplo de salida de su ejecución:

```
The mouse Fievel has started to feed
The mouse Jerry has started to feed
The mouse Pinky has started to feed
The mouse Mickey has started to feed
The mouse Pinky has stopped to feed
The mouse Fievel has stopped to feed
The mouse Jerry has stopped to feed
The mouse Mickey has stopped to feed
```

Todos los ratones han comenzado a alimentarse de inmediato, sin esperar a que termine ninguno de los demás. El tiempo total del proceso será, aproximadamente, el tiempo del proceso más lento (en este caso, 6 segundos). La reducción del tiempo total de ejecución es evidente.

5.3. Ejemplo03

Retomando el enunciado del ejemplo de los objetos de la clase Raton, la solución mediante implementación de la interface Runnable tendría el código que se muestra a continuación, revisa las partes del código que se han modificado respecto de la solución anterior en la que se heredaba de Thread.

```java
public class Mouse implements Runnable {

    private String name;
    private int feedingTime;

    public Mouse(String name, int feedingTime) {
        super();
        this.name = name;
        this.feedingTime = feedingTime;
    }

    public void eat() {
        try {
            System.out.printf("The mouse %s has started to feed%n", name);
            Thread.sleep(feedingTime * 1000);
            System.out.printf("The mouse %s has stopped to feed%n", name);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void run() {
        this.eat();
    }

    public static void main(String[] args) {
        Mouse fievel = new Mouse("Fievel", 4);
        Mouse jerry = new Mouse("Jerry", 5);
```

```
        Mouse pinky = new Mouse("Pinky", 3);
        Mouse mickey = new Mouse("Mickey", 6);
        new Thread(fievel).start();
        new Thread(jerry).start();
        new Thread(pinky).start();
        new Thread(mickey).start();
    }
}
```

El resultado de ejecución:

```
The mouse Fievel has started to feed
The mouse Pinky has started to feed
The mouse Jerry has started to feed
The mouse Mickey has started to feed
The mouse Pinky has stopped to feed
The mouse Fievel has stopped to feed
The mouse Jerry has stopped to feed
The mouse Mickey has stopped to feed
```

Se puede observar que el código es muy similar, salvo en la declaración de la clase (implementación de una interface frente a herencia) y en la creación de los hilos y su arranque: los objetos `Runnable` deben «envolverse» en objetos `Thread` para poder ser arrancados. Es importante apreciar que en ambos casos la ejecución se realiza invocando al método `start`.

5.4. Ejemplo04

Ejemplo multihilo a partir de un único.

```
public class SimpleMouse implements Runnable {

    private String name;
    private int feedingTime;
    private int consumedFood;

    public SimpleMouse(String name, int feedingTime) {
        super();
        this.name = name;
        this.feedingTime = feedingTime;
    }

    public void eat() {
        try {
            System.out.printf("The mouse %s has started to feed%n", name);
            Thread.sleep(feedingTime * 1000);
            consumedFood++;
            System.out.printf("The mouse %s has stopped to feed%n", name);
            System.out.printf("Consumed food: %d%n", consumedFood);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
    }

    @Override
    public void run() {
        this.eat();
    }

    public static void main(String[] args) {
        SimpleMouse fievel = new SimpleMouse("Fievel", 4);
        new Thread(fievel).start();
        new Thread(fievel).start();
        new Thread(fievel).start();
        new Thread(fievel).start();
    }
}
```

El resultado de la ejecución es el siguiente:

```
The mouse Fievel has started to feed
The mouse Fievel has started to feed
The mouse Fievel has started to feed
The mouse Fievel has started to feed
The mouse Fievel has stopped to feed
The mouse Fievel has stopped to feed
Consumed food: 3
The mouse Fievel has stopped to feed
Consumed food: 2
Consumed food: 4
The mouse Fievel has stopped to feed
Consumed food: 4
```

Cada hilo ha ejecutado el método `run` sobre los datos del mismo objeto. Es decir, se ha ejecutado simultáneamente cuatro veces un bloque de código de un único objeto, compartiendo sus atributos. De esta forma, en la salida se puede apreciar que el valor del atributo `consumedFood` se ha incrementado en 1 por cada hilo. Esto se puede observar porque el valor de `consumedFood` se ha incrementado varias veces durante la ejecución. El hecho de que algunos valores intermedios no aparezcan en la siguiente captura de la salida de la ejecución tiene que ver con la `asincronía` y el alto coste de ejecución que tienen las sentencias que escriben por pantalla.

Aunque se tratará más adelante en esta misma unidad, es importante insistir en que los diferentes hilos del ejemplo anterior están trabajando sobre una única copia del objeto en memoria, por lo que las variables (los atributos) son compartidas, pudiendo sufrir errores de concurrencia.

5.5. Ejemplo04bis

Por ejemplo, sustituyendo el método main del ejemplo anterior por el siguiente código, se pueden crear y ejecutar varios threads mediante un bucle for a partir de una única instancia de una clase que implementa la interfaz `Runnable`. El resultado, con un número bajo de iteraciones (por ejemplo, 4) será habitualmente correcto (el atributo `consumedFood` alcanzará el mismo valor que el número de iteraciones). Con un número alto (por ejemplo, 1000 iteraciones) el resultado será habitualmente erróneo (el atributo `consumedFood` alcanzará un valor por debajo del número de iteraciones). Esto se debe a que al compartir todos los hilos los mismos atributos producen errores de concurrencia que deben evitarse mediante técnicas concretas que veremos más adelante.

```
public static void main(String[] args) {
    SimpleMouse fievel = new SimpleMouse("Fievel", 4);
    for (int i = 0; i < 1000; i++) {
        new Thread(fievel).start();
    }
}
```

En una ejecución del código anterior, el resultado es 998, cuando debería haber sido 1000 si no hubiese habido problemas de concurrencia

```
Consumed food: 998
```

5.6. Ejemplo05

Listado de estados por los que pasa un hilo:

```
import java.util.ArrayList;

public class SimpleMouse implements Runnable {

    private String name;
    private int feedingTime;
    private int consumedFood;

    public SimpleMouse(String name, int feedingTime) {
        super();
        this.name = name;
        this.feedingTime = feedingTime;
    }

    public void eat() {
        try {
            System.out.printf("The mouse %s has started to feed%n", name);
            Thread.sleep(feedingTime * 1000);
            consumedFood++;
            System.out.printf("The mouse %s has stopped to feed%n", name);
            System.out.printf("Consumed food: %d%n", consumedFood);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
    }

    @Override
    public void run() {
        this.eat();
    }

    public static void main(String[] args) {
        SimpleMouse mickey = new SimpleMouse("Mickey", 6);
        ArrayList<Thread.State> threadState = new ArrayList();
        Thread h = new Thread(mickey);
        threadState.add(h.getState());
        h.start();

        while (h.getState() != Thread.State.TERMINATED) {
            if (!threadState.contains(h.getState())) {
                threadState.add(h.getState());
            }
        }
        if (!threadState.contains(h.getState())) {
            threadState.add(h.getState());

        }
        for (Thread.State estado : threadState) {
            System.out.println(estado);
        }
    }
}
```

Al finalizar la ejecución se muestran los estados recogidos:

```
The mouse Mickey has started to feed
The mouse Mickey has stopped to feed
Consumed food: 1
NEW
RUNNABLE
TIMED_WAITING
TERMINATED
```

5.7. Ejemplo06

En el siguiente ejemplo se muestra un uso conjunto de las clases `Timer` y `TimerTask`. El programa simula controlar un sistema de regadío automático. Este sistema riega por primera vez transcurridos `1000` milisegundos desde el inicio de la ejecución y repite el riego cada `2000` milisegundos.

```
import java.util.Timer;
import java.util.TimerTask;

public class IrrigationSystem extends TimerTask {

```

```
    @Override
    public void run() {
        System.out.println("Watering...");
    }

    public static void main(String[] args) {
        Timer timer = new Timer();
        timer.schedule(new IrrigationSystem(),1000,2000);
    }
}
```

La salida generada transcurridos unos segundos es la que se muestra a continuación. Entre la escritura de cada línea transcurren 2000 milisegundos.

```
Watering...
Watering...
Watering...
...
```

5.8. Ejemplo07

El siguiente ejemplo ilustra el uso de `ScheduledExecutorService` como herramienta para la programación de ejecuciones de tareas. Como se puede observar, mediante la clase `Executors` se obtiene una instancia de `ScheduledExecutorService`, que es la interface que permite programar tareas recurrentes en hilos independientes. Una vez obtenido el programador de tareas, se le indica qué tarea se quiere ejecutar (objeto `sr`), cuántas unidades de tiempo se desea esperar hasta que se inicie la primera tarea (1), cuántas unidades de tiempo se desea esperar entre cada repetición de la tarea (2) y en qué unidad están representadas las unidades de tiempo (`TimeUnit.SECONDS`)

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class IrrigationSystem implements Runnable {

    @Override
    public void run() {
        System.out.println("Watering...");
    }

    public static void main(String[] args) {
        IrrigationSystem ir = new IrrigationSystem();
        ScheduledExecutorService ses = Executors.newSingleThreadScheduledExecutor();
        ses.scheduleAtFixedRate(ir, 1, 2, TimeUnit.SECONDS);
        System.out.println("Configured irrigation system");
    }
}
```

La salida de la ejecución transcurridos unos segundos es la que se muestra a continuación. Entre cada escritura del texto «Regando» transcurren dos segundos.

```
Configured irrigation system
Watering...
Watering...
```

5.9. Ejemplo08

Para ilustrar mejor el comportamiento de este tipo de soluciones se presenta el siguiente ejemplo. utiliza Una cola recibe escrituras y lecturas de un conjunto de hilos. La primera solución utiliza una estructura de datos `LinkedList` como un soporte. Al no ser esta estructura segura frente a múltiples hilos, la ejecución produce un error.

```java
import java.util.LinkedList;
import java.util.Queue;

public class nonConcurrentQueue implements Runnable {

    private static Queue<Integer> queue = new LinkedList<Integer>();

    @Override
    public void run() {
        queue.add(10);
        for (Integer i : queue) {
            System.out.print(1 + ":");
        }
        System.out.println("Queue size:" + queue.size());
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread(new nonConcurrentQueue()).start();
        }
    }
}
```

La salida será similar a esta:

```
Exception in thread "Thread-1" Exception in thread "Thread-0"
java.util.ConcurrentModificationException
1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:Queue size:6
Queue size:4
Queue size:5
Queue size:3
1:1:1:1:1:1:1:Queue size:7
1:1:1:1:1:1:1:1:Queue size:8
1:1:1:1:1:1:1:1:1:Queue size:9
1:1:1:1:1:1:1:1:1:1:Queue size:10
    at java.base/java.util.LinkedList$ListItr.checkForComodification(LinkedList.java:970)
[...]
```

5.10. Ejemplo08bis

La segunda solución utiliza el mismo código cambiando la estructura de datos a `ConcurrentLinkedDeque` obteniendo una ejecución sin errores.

```java
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedDeque;

public class ConcurrentQueue implements Runnable {

    private static Queue<Integer> queue = new ConcurrentLinkedDeque<Integer>();

    @Override
    public void run() {
        queue.add(10);
        for (Integer i : queue) {
            System.out.print(1 + ":");
        }
        System.out.println("Queue size:" + queue.size());
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Thread(new ConcurrentQueue()).start();
        }
    }
}
```

Salida obtenida:

```
1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:Queue size:2
Queue size:4
Queue size:2
Queue size:7
Queue size:7
Queue size:7
Queue size:3
1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:Queue size:9
1:Queue size:9
1:1:1:1:1:1:1:1:1:1:Queue size:10
```

Aunque de esta última ejecución no está ordenada debido a la concurrencia de los hilos, sí se puede observar que la ejecución ha generado una cola con 10 elementos correspondientes a los 10 hilos que estaban accediendo a leer y escribir en la estructura de datos.

5.11. Ejemplo09

En el siguiente ejemplo se muestra el uso de `Exchanger`. Se crean dos clases que implementan `Runnable`, `TaskA` y `TaskB`. Ambas clases reciben una instancia de `Exchanger` en el constructor y la utilizan para intercambiar información entre sí. La llamada al método `exchange` por parte de una de las dos tareas producirá un bloqueo de espera hasta que la otra tarea haga lo propio, intercambiando en ese momento la información entre los dos hilos. Por su parte, la clase `Commuter`, construye tanto el objeto `Exchanger` como las dos tareas programadas en `TaskA` y `TaskB`. Es importante prestar atención al hecho de que las tareas no tienen referencias la una de la otra, sino que tienen acceso al objeto que hace de «intercambiador» de información.

`TaskA`:

```java
import java.util.concurrent.Exchanger;

public class TaskA implements Runnable {

    private Exchanger<String> exchanger;

    public TaskA(Exchanger<String> exchanger) {
        super();
        this.exchanger = exchanger;
    }

    @Override
    public void run() {
        try {
            String receivedMessage = exchanger.exchange("Message sent by TaskA");
            System.out.println("Message received in TaskA: " + receivedMessage);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

`TaskB`:

```
import java.util.concurrent.Exchanger;

public class TaskB implements Runnable {

    private Exchanger<String> exchanger;

    public TaskB(Exchanger<String> exchanger) {
        super();
        this.exchanger = exchanger;
    }

    @Override
    public void run() {
        try {
            String receivedMessage = exchanger.exchange("Message sent by TaskB");
            System.out.println("Message received in TaskB: " + receivedMessage);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

`Commuter`:

```
import java.util.concurrent.Exchanger;

public class Commuter {

    public static void main(String[] args) {
        Exchanger<String> exchanger = new Exchanger<String>();
        new Thread(new TaskA(exchanger)).start();
        new Thread(new TaskB(exchanger)).start();
    }
}
```

La salida obtenida será:

```
Message received in TaskA: Message sent by TaskB
Message received in TaskB: Message sent by TaskA
```

5.12. Ejemplo10

```
import java.util.ArrayList;
import java.util.List;

public class nonSafeReaderWriter extends Thread {
```

```
5
6       private static List<String> words = new ArrayList<String>();
7
8       @Override
9       public void run() {
10          words.add("New word");
11          for (String word : words) {
12              words.size();
13          }
14          System.out.println(words.size());
15      }
16
17      public static void main(String[] args) {
18          for (int i = 0; i < 100; i++) {
19              new nonSafeReaderWriter().start();
20          }
21      }
22  }
```

En la salida aparecen referencias a excepciones que indican que han ocurrido errores de concurrencia en el acceso a la lista, como `java.util.ConcurrentModificationException`:

```
1  [...]
2  Exception in thread "Thread-21" java.util.ConcurrentModificationException
3  [...]
```

5.13. Ejemplo11

```
1   import java.util.List;
2   import java.util.concurrent.CopyOnWriteArrayList;
3
4   public class safeReaderWriter extends Thread {
5
6       private static List<String> words = new CopyOnWriteArrayList<String>();
7
8       @Override
9       public void run() {
10          words.add("New word");
11          for (String word : words) {
12              words.size();
13          }
14          System.out.println(words.size());
15      }
16
17      public static void main(String[] args) {
18          for (int i = 0; i < 100; i++) {
19              new safeReaderWriter().start();
20          }
21      }
22  }
```

```
23
```

Que esta vez se ejecuta sin problemas ni errores.

5.14. Ejemplo12

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Reader implements Callable<String> {

    @Override
    public String call() throws Exception {
        String readedText = "I like action movies";
        Thread.sleep(1000);
        return readedText;
    }

    public static void main(String[] args) {
        try {
            Reader reader = new Reader();
            ExecutorService executionService = Executors.newSingleThreadExecutor();
            Future<String> result = executionService.submit(reader);
            String text = result.get();
            if (result.isDone()) {
                System.out.println(text);
                System.out.println("Process finished");
            } else if (result.isCancelled()) {
                System.out.println("Process cancelled");
            }
            executionService.shutdown();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

La salida producida es la siguiente:

```
Me gustan las películas de acción
Proceso finalizado
```

Como se ha indicado anteriormente, la principal diferencia entre implementar `Callable` y `Runnable` es que la primera opción puede proporcionar un retorno. No obstante, con `Runnable` existen técnicas para transmitir valores mediante el uso de referencias a objetos.

5.15. Ejemplo13

```java
public class BasicRunnable implements Runnable {

    private int id;

    public BasicRunnable(int id) {
        super();
        this.id = id;
    }

    @Override
    public void run() {
        try {
            System.out.println("Processing thread " + id);
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            BasicRunnable br = new BasicRunnable(i);
            new Thread(br).start();
        }
    }
}
```

La salida generada para una ejecución cualquiera será similar a esta:

```
Processing thread 9
Processing thread 1
Processing thread 5
Processing thread 6
Processing thread 3
Processing thread 7
Processing thread 8
Processing thread 4
Processing thread 2
Processing thread 0
```

Como se puede observar, el orden de las escrituras no coincide con el orden de las ejecuciones de los hilos.

5.16. Ejemplo14

```java
public class BasicRunnableLambda {

    private int id;

```

```
    public BasicRunnableLambda(int id) {
        super();
        this.id = id;
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            BasicRunnableLambda br = new BasicRunnableLambda(i);
            new Thread(() -> {
                try {
                    System.out.println("Processing thread " + br.id);
                    Thread.sleep(10000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }).start();
        }
    }
}
```

La salida no difiere del ejemplo anterior.

5.17. Ejemplo15

```
public class BasicThread extends Thread{
        private int id;

    public BasicThread(int id) {
        super();
        this.id = id;
    }

    @Override
    public void run() {
        try {
            System.out.println("Processing thread " + id);
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            BasicThread bt = new BasicThread(i);
            new Thread(bt).start();
        }
    }

}
```

La salida no difiere de la clase que implementamos anteriormente con `Runnable`.

5.17.1. Ejemplo16

```
public class BasicInterruption extends Thread {

    @Override
    public void run() {
        int counter = 0;
        while (true) {
            counter++;
            try {
                System.out.println(counter);
                if (counter == 3) {
                    System.out.print("Interruption");
                    this.interrupt();
                }
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                return;
            }
        }

    }

    public static void main(String[] args) {
        new BasicInterruption().start();
    }
}
```

La salida al ejecutar el código debe ser:

```
1
2
3
Interruption
```

5.18. Ejemplo17

SharedObject class:

```
public class SharedObject {
    public int sharedVariable;
}
```

UniqueInstanceSharing class:

```
public class UniqueInstanceSharing extends Thread {

```

```
    private SharedObject so;

    public UniqueInstanceSharing(SharedObject so) {
        this.so = so;
    }

    @Override
    public void run() {
        this.so.sharedVariable++;
        System.out.println("Shared Variable: " + this.so.sharedVariable);
    }

    public static void main(String[] args) throws InterruptedException {
        SharedObject so = new SharedObject();
        UniqueInstanceSharing uis1 = new UniqueInstanceSharing(so);
        UniqueInstanceSharing uis2 = new UniqueInstanceSharing(so);
        uis1.start();
        Thread.sleep(1000);
        uis2.start();
    }
}
```

La salida será similar a esta:

```
Shared Variable: 1
Shared Variable: 2
```

5.19. Ejemplo18

```
public class RunnableSharing extends Thread {

    private int counter;

    @Override
    public void run() {
        counter++;
        System.out.println("Counter: " + counter);
    }

    public static void main(String[] args) throws InterruptedException {
        RunnableSharing rs = new RunnableSharing();
        for (int i = 0; i < 1000; i++) {
            new Thread(rs).start();
        }
    }
}
```

Debes obtener una salida similar a esta:

```
[...]
Counter: 998
Counter: 999
Counter: 1000
```

En este caso aunque llamemos más de 1000 veces no existen problemas al acceder a la variable `counter`.

5.20. Ejemplo19

En este ejemplo 1000 hilos incrementan en 1000 unidades una variable estática común. La variable debería obtener como resultado 1000000.

```
public class SharedVariable extends Thread {

    private static int counter;

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter++;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 1000; i++) {
            new SharedVariable().start();
        }
        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("Counter value: " + counter);
    }
}
```

Obtendremos una salida similar a esta:

```
Counter value: 993203
```

Que dista bastante del valor esperado. Esto es debido a que la operación de incremento con el operador unario ++ no realiza una operación atómica (que se ejecuta sin interrupciones), sino que consta de varios pasos y la ejecución se puede interrumpir en cualquiera de ellos, provocando un problema de inconsistencia de memoria.

Suponiendo que el proceso de incremento en 1 de una variable está compuesto de los siguientes pasos:

- Lectura del valor de la variable.
- Incremento en 1 del valor de la variable.
- Escritura del valor de la variable.

Si un hilo lee el valor de la variable siendo este 2, calcula el nuevo valor y antes de escribir el 3 resultante otro hilo toma su lugar en el procesador, este leerá de nuevo 2, calculará el nuevo valor que será 3 y lo escribirá, cuando el primer hilo vuelva al procesador continuará ejecutando el tercer paso y escribiendo el valor 3. Dos hilos han incrementado en 1 el valor de una misma variable, pero al finalizar solo se ha reflejado uno de los incrementos. Esto explica las pérdidas de incrementos que se muestran en el ejemplo.

5.21. Ejemplo20

```
1  public class SimpleWaitNotify implements Runnable {
2
3      private volatile boolean runningMethod1 = false;
4
5      public synchronized void method1() {
6          for (int i = 0; i < 10; i++) {
7              System.out.printf("Method1: Running %d\n", i);
8              if (i == 5) {
9                  try {
10                     this.wait();
11                 } catch (InterruptedException e) {
12                     e.printStackTrace();
13                 }
14             }
15         }
16     }
17
18     public synchronized void method2() {
19         for (int i = 10; i < 20; i++) {
20             System.out.printf("Method2: Running %d\n", i);
21         }
22         this.notifyAll();
23     }
24
25     @Override
26
27     public void run() {
28         if (!runningMethod1) {
29             runningMethod1 = true;
30             method1();
31         } else {
32             method2();
33         }
34     }
35
36     public static void main(String[] args) {
37         SimpleWaitNotify swn = new SimpleWaitNotify();
```

```
        new Thread(swn).start();
        new Thread(swn).start();
    }
}
```

La salida debe ser:

```
Running 0
Running 1
Running 2
Running 3
Running 4
Running 5
Running 10
Running 11
Running 12
Running 13
Running 14
Running 15
Running 16
Running 17
Running 18
Running 19
Running 6
Running 7
Running 8
Running 9
```

5.22. Ejemplo21

```
public class Basic extends Thread {

    private int id;

    public Basic(int id) {
        this.id = id;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < 3; i++) {
                System.out.printf("Thread %d. Interaction %d\n", id, i);
                Thread.sleep(1000);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

```

```
    public static void main(String[] args) {
        Basic thread1 = new Basic(1);
        Basic thread2 = new Basic(2);
        thread1.start();
        thread2.start();
    }
}
```

5.23. Ejemplo22

```
public class BasicJoin extends Thread {

    private int id;
    private boolean suspend = false;
    private Thread referenceThread;

    public BasicJoin(int id) {
        this.id = id;
    }

    public void threadSuspend(Thread referenceThread) {
        this.suspend = true;
        this.referenceThread = referenceThread;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < 3; i++) {
                if (suspend) {
                    referenceThread.join();
                }
                System.out.printf("Thread %d. Interaction %d\n", id, i);
                Thread.sleep(1000);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        BasicJoin thread1 = new BasicJoin(1);
        BasicJoin thread2 = new BasicJoin(2);
        thread1.start();
        thread2.start();
        thread2.threadSuspend(thread1);
    }
}
```

5.24. Ejemplo23

```java
public class MethodSyncronization implements Runnable {

    public synchronized void method1() {
        System.out.println("Method 1 start");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ie) {
            return;
        }
        System.out.println("Method 1 ends");
    }

    public synchronized void method2() {
        System.out.println("Method 2 start");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ie) {
            return;
        }
        System.out.println("Method 2 ends");
    }

    @Override
    public void run() {
        method1();
        method2();
    }

    public static void main(String[] args) {
        MethodSyncronization ms = new MethodSyncronization();
        new Thread(ms).start();
        new Thread(ms).start();
    }
}
```

Los métodos se ejecutarán de uno en uno, pese a que se dispone de dos hilos distintos. La salida generada es la siguiente:

```
Method 1 start
Method 1 ends
Method 2 start
Method 2 ends
Method 1 start
Method 1 ends
Method 2 start
Method 2 ends
```

Recuerda que el orden puede variar, lo importante es que los métodos se ejecutan todos unos detrás de otros.

Si los métodos no estuviesen sincronizados (`MethodSyncronizationBis`) los dos hilos ejecutarían simultáneamente el primer método, tras su finalización, el segundo. La salida habría resultado la siguiente:

```
Method 1 start
Method 1 start
Method 1 ends
Method 1 ends
Method 2 start
Method 2 start
Method 2 ends
Method 2 ends
```

5.25. Ejemplo24

```
public class WrongMethodSyncronization extends Thread {

    public synchronized void method1() {
        System.out.println("Method 1 start");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ie) {
            return;
        }
        System.out.println("Method 1 ends");
    }

    public synchronized void method2() {
        System.out.println("Method 2 start");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ie) {
            return;
        }
        System.out.println("Method 2 ends");
    }

    @Override
    public void run() {
        method1();
        method2();
    }

    public static void main(String[] args) {
        new WrongMethodSyncronization().start();
        new WrongMethodSyncronization().start();
    }
}
```

La salida en este caso será:

```
Method 1 start
Method 1 start
Method 1 ends
Method 2 start
Method 1 ends
Method 2 start
Method 2 ends
Method 2 ends
```

5.26. Ejemplo25

```
public class SegmentSyncronization extends Thread {

    int id;
    static Object block1 = new Object();
    static Object block2 = new Object();

    public SegmentSyncronization(int id) {
        this.id = id;
    }

    public void method1() {
        synchronized (block1) {
            System.out.printf("Method 1 from thread %d start\n", id);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) {
                return;
            }
            System.out.printf("Method 1 from thread %d ends\n", id);
        }
    }

    public void method2() {
        synchronized (block2) {
            System.out.printf("Method 2 from thread %d start\n", id);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) {
                return;
            }
            System.out.printf("Method 2 from thread %d ends\n", id);
        }
    }

    @Override
    public void run() {
        if (id == 1) {
            method1();
            method2();
```

```
        } else {
            method2();
            method1();
        }
    }

    public static void main(String[] args) {
        new SegmentSyncronization(1).start();
        new SegmentSyncronization(2).start();
    }
}
```

La salida será similar a esta:

```
Method 1 from thread 1 start
Method 2 from thread 2 start
Method 1 from thread 1 ends
Method 2 from thread 2 ends
Method 2 from thread 1 start
Method 1 from thread 2 start
Method 2 from thread 1 ends
Method 1 from thread 2 ends
```

5.27. Ejemplo26

```
package UD02.Example26;

import java.util.concurrent.Semaphore;

public class BasicSemaphore implements Runnable {

    Semaphore semaphore = new Semaphore(3);

    @Override
    public void run() {
        try {
            semaphore.acquire();
            System.out.println("Step 1");
            Thread.sleep(1000);
            System.out.println("Step 2");
            Thread.sleep(1000);
            System.out.println("Step 3");
            Thread.sleep(1000);
            semaphore.release();
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
    }

    public static void main(String[] args) {
```

```
        BasicSemaphore sb = new BasicSemaphore();
        for (int i = 0; i < 5; i++) {
            new Thread(sb).start();
        }
    }
}

```

Salida:

```
Step 1
Step 1
Step 1
Step 2
Step 2
Step 2
Step 3
Step 3
Step 3
Step 1
Step 1
Step 2
Step 2
Step 3
Step 3
```

La salida muestra que los tres primeros hilos que han intentado acceder a la sección crítica a través del bloqueo del semáforo se han ejecutado concurrentemente, y el resto de los hilos se han ejecutado cuando los primeros han terminado y liberado los bloqueos.

6. Fuentes de información

- Wikipedia
- Programación de servicios y procesos - FERNANDO PANIAGUA MARTÍN [Paraninfo]
- Programación de Servicios y Procesos - ALBERTO SÁNCHEZ CAMPOS [Ra-ma]
- Programación de Servicios y Procesos - Mª JESÚS RAMOS MARTÍN - [Garceta] (1ª y 2ª Edición)
- Programación de servicios y procesos - CARLOS ALBERTO CORTIJO BON [Sintesis]
- Programació de serveis i processos - JOAR ARNEDO MORENO, JOSEP CAÑELLAS BORNAS i JOSÉ ANTONIO LEO MEGÍAS [IOC]
- GitHub repositories:
    - https://github.com/ajcpro/psp
    - https://oscarmaestre.github.io/servicios/index.html
    - https://github.com/juanro49/DAM/tree/master/DAM2/PSP
    - https://github.com/pablohs1986/dam_psp2021
    - https://github.com/Perju/DAM
    - https://github.com/eldiegoch/DAM
    - https://github.com/eldiegoch/2dam-psp-public
    - https://github.com/franlu/DAM-PSP
    - https://github.com/ProgProcesosYServicios
    - https://github.com/joseluisgs
    - https://github.com/oscarnovillo/dam2_2122
    - https://github.com/PacoPortillo/DAM_PSP_Tarea02_La-Cena-de-los-Filosofos